CS 610 Semester 2022–2023-I: Assignment 2

21st August 2020

Due Your assignment is due by Sep 3, 2022, 11:59 PM IST.

General Policies

- You should do this assignment ALONE.
- Do not plagiarize or turn in solutions from other sources. You will be PENALIZED if caught.

Submission

- Submission will be through mooKIT.
- Submit a PDF file with name "<roll-no>.pdf". Include the solution to Problem 1 in the PDF, and explain your implementations, results, and other issues (if any) for Problems 2–4. We encourage you to use the LaTeX typesetting system for generating the PDF file.
- Name each source file for Problems 2–4 "<roll-no-probno>.cpp" where "probno" stands for the problem number (e.g., "22111045-prob2.cpp").
- You will get up to TWO LATE days to submit your assignment, with a 25% penalty for each day.

Evaluation

- Your solutions should only use concepts that have been discussed in class.
- Show your computations where feasible and justify briefly.
- Write your code such that the EXACT output format is respected.
- We will evaluate the implementations with our OWN inputs and test cases, so remember to test thoroughly.
- We may deduct marks if you disregard the listed rules.

Problem 1 **[10 marks]**

Consider the following loop nest.

```
for i = 1, N-2
 for j = i+1, i+N-2
  A(i, i-j) = A(i, i-j-1) - A(i+1, i-j) + A(i-1, i-j-1)
```

List all flow, anti, and output dependences, if any, using Delta test. Assume all array subscript references are valid.

Problem 2

[20 marks]

You are given an integer array *nums* of length n which represents a random permutation of all the integers in the range $[0, n-1]$. The number of inversions is the number of different pairs (i, j) where $0 <= i < j < n$, such that $nums[i] > nums[j]$. Count the total number of inversions in parallel using p threads.

Your implementation should take two command line parameters: n and p. You can adapt the following snippet to initialize the array *nums* with random integers in the range $[0, n-1]$.

```
// Generate "max" random numbers in the range [0, max-1] and
// fill in the output variable "array"
void generate_rand_nums(int max, int *array) {
 const int range_from = 0;
 const int range_to = max - 1;
 std::random_device rd; // obtain a random number from hardware
 std::mt19937 gen(rd()); // seed the generator
 std::uniform_int_distribution<int> distr(range_from, range_to); // inclusive

 for (int i = 0; i < max; i++) {
  array[i] = distr(gen); // generate numbers
 }
}
```

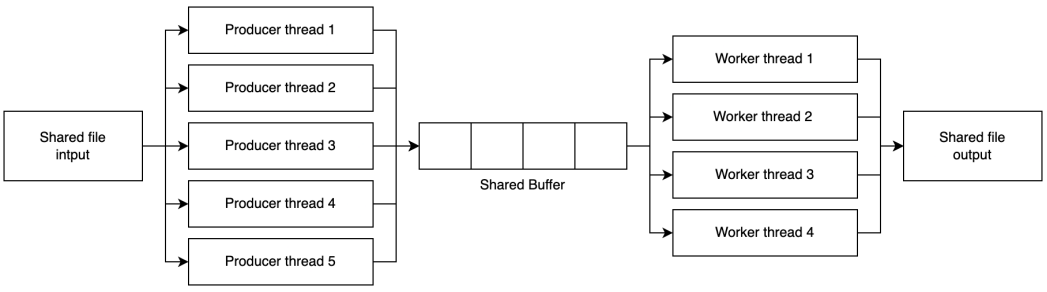
Problem 3

[50 marks]

The producer and consumer problem can be modeled as a synchronization problem. We will practice implementing one scenario using Pthreads. You are free to use any Pthread feature.

Consider an input file "input.txt" that contains integer numbers, each number is on a separate line. There are `P` producer threads. Each producer thread reads a number from a new line in the input file and writes that number into a shared buffer. The shared buffer can hold up to `B` numbers. There are `W` worker threads that pick a number from the shared buffer (i.e., the slot becomes empty) and check whether the number is prime. If the number is not prime, the worker thread does nothing. If the number is prime, the worker thread writes the number on a new line in a text file called "prime.txt". All the worker threads write to the same shared output file "prime.txt".

Your implementation should avoid synchronization problems and should take three command line parameters: `P`, `B`, and `W`.



Take care of the following points.

- Two producer threads should not read a number from the same line in the input file.

- A number written to the shared buffer by a producer thread should not be overwritten by other threads.
- Two worker threads should not read the same number from the same slot of the shared buffer.
- A worker thread should not overwrite a prime number written by another worker thread in the file "prime.txt".
- A producer thread blocks until there is an empty slot in the shared buffer to write.
- A worker thread blocks until there is a non-empty slot in the shared buffer that can be read.
- The implementation will run till there are no more numbers in "input.txt". When the input file is processed, one or more producer threads will signal the worker threads that there is no more new work and the worker threads can terminate after the current batch of work is processed.

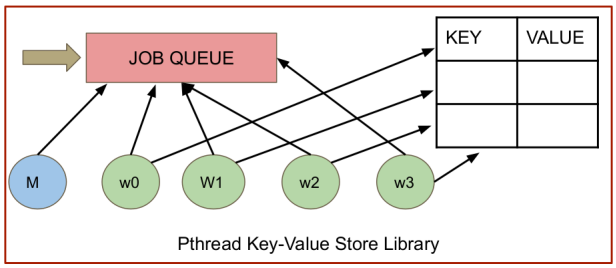
Problem 4 **[50 marks]**

UPDATE: The following description has been revised, and hopefully simplified, on Sep 3, 2022. As a result, we have extended the submission deadline to 11:59 PM on Sep 5, 2022. All other policies remain the same.

You need to implement a multithreaded key-value store library using Pthreads. The store library should support Insert(key, value), Update(key, value), Delete(key), and Find(key) operations on the key-value store. Remember that the functions can be called in parallel by concurrent threads.

The following figure shows details of the key-value store. The library consists of four worker threads (W0, W1, W2, and W3) and a manager thread (M) that is basically the main thread. There is a "Job Queue" which holds operations that need to be executed on the key-value store by the worker threads. These operations are Insert(), Update(), Delete(), and Find(). Note that multiple operations may be executed in parallel by worker threads on the key-value store.

The manager thread enqueues operations (e.g., call to Insert()) in the Job Queue. The worker threads consume operations from the front of the Job Queue and perform them on the key-value store. You should allow fine-grained concurrency between picking up jobs from the Job Queue and executing them on the key-value store. That is, your implementation should avoid using a global lock on both the Job Queue and the key-value store. This will potentially allow thread W0 to execute a dequeued operation on the key-value store while a second thread W3 may start consuming the next available data from the Job Queue.



The library provides the following APIs for client programs.

- `int enqueue (struct operation *op)`: This creates an entry in the Job Queue by the manager thread.

```
struct operation {
 uint8_t type; // 0 - insert, 1 - update, 2 - delete, 3 - find
 uint32_t key;
 uint64_t value;
};
```

Attributes `type` specifies the operation type and key, `value` specifies the corresponding (key,value) pair for an Insert or Update operation. The `value` field is irrelevant for Delete and Find operations.

On success, `enqueue` returns the location (between 0 and *max* queue size - 1) in the Job Queue. The `enqueue` implementation is non-blocking; it returns -1 when the Job Queue is full. Furthermore, it returns -1 on when the `key` for a Find() operation is not present.

Design constraints:

- The dynamic size of the key-value store will depend on insert/delete operations.
- The size of the Job Queue is 8.
- Your implementation should take one command line parameter, n, that specifies the total number of operations enqueued by the manager thread to the Job Queue.
- The library should ensure mutual exclusion when (i) worker threads operate at the same location in the key-value store, and (ii) manager and worker threads when performing enqueue and dequeue operations on the Job Queue.

You are free to use any C, C++, and Pthread features. Feel free to modify the shared template code according to the revised problem description.